Ulrich Flegel, Michael Meier (Eds.)

# Detection of Intrusions and Malware & Vulnerability Assessment

GI Special Interest Group SIDAR Workshop, DIMVA 2004
Dortmund, Germany, July 6-7, 2004
Proceedings


Gesellschaft für Informatik 2004

**Lecture Notes in Informatics (LNI) - Proceedings**
Series of the Gesellschaft für Informatik (GI)

Volume P-46

ISBN 3-88579-375-X
ISSN 1617-5468

**Volume Editors**
Ulrich Flegel
University of Dortmund,
Computer Science Department, Chair VI, ISSI
D-44221 Dortmund, Germany
ulrich.flegel@udo.edu
Michael Meier
Brandenburg University of Technology Cottbus,
Computer Science Department, Chair Computer Networks
P.O. Box 10 13 44, D-03013 Cottbus, Germany
mm@informatik.tu-cottbus.de


**printed by** Köllen Druck+Verlag GmbH, Bonn

# Structural Comparison of Executable Objects

Halvar Flake

halvar@blackhat.com

**Abstract:** A method to heuristically construct an isomorphism between the sets of functions in two similar but differing versions of the same executable file is presented. Such an isomorphism has multiple practical applications, specifically the ability to detect programmatic changes between the two executable versions. Moreover, information (function names) which is available for one of the two versions can also be made available for the other .

A framework implementing the described methods is presented, along with empirical data about its performance when used to analyze patches to recent security vulnerabilities. As a more practical example, a security update which fixes a critical vulnerability in an H.323 parsing component is analyzed, the relevant vulnerability extracted and the implications of the vulnerability and the fix discussed.

## 1 Introduction

While programs that compare different versions of the same source code file have been in widespread use for many years, very little focus has so far been placed on the importance of detecting and analyzing changes between two versions of the same executable.

Without an automated way of detecting source code changes in the object code resulting from compilation, the party prompted with the task of reverse engineering the changes from the object code is at a disadvantage: It takes relatively little work to change source code and recompile, while the analysis of the object code will have to be completely redone to detect the changes. Both virus authors of high-level-language virus families (such as SoBig) and closed-source software vendors try to exploit this asymmetry: The authors of SoBig intend to create large quantities of work for the antivirus researchers to have more time to use the infrastructure built by their worm, whereas closed source vendors hope their customers will have time for installing patches because possible attackers presumably need a lot of time to reverse-engineer the relevant changes from object-code-only security updates.

This paper presents a novel approach which corrects the abovementioned asymmetry: Given two variants of the same executable $A$ called $A'$ and $A''$, an one-to-one mapping between all the functions in $A'$ to all functions in $A''$ is created. The mapping does not depend on the specific assembly-level instructions generated by the compiler but is more general in nature: It maps *control flow graphs* of functions, thus ignoring less-aggressive optimization such as instruction reordering and changes in register allocation.

This allows porting of information (such as function names from symbolic debug information or prior analysis) from one executable to another. Furthermore, due to the approach taken, functions that have changed their functionality significantly will not be mapped, allowing the easy detection of functional changes to the program.

Detecting programmatic changes between two versions of the same executable is relevant to security research as it allows for quick analysis of security updates (”patches”) to extract detailed information about the underlying security vulnerabilities. This allows for quick assessment of the risk posed by a particular problem and can be used to prevent vendors from fixing security issues ”silently”, e.g. without notifying their customers about the security problem.

## 2 Previous Work

Automatically analyzing and classifying changes to source code have been studied extensively in literature before, and listing all relevant papers seems to be out of scope for this paper. Most of this research focuses on treating the source code as a sequence of lines, and applying a sequence-comparison algorithm [Hir77][HS77].

The problem of matching functions in two executables to form pairs has been studied in [ZW00, ZW99], although focused on reuse of profiling information which allowed the assumption of symbols for both executables being available. Other work has been done with focus on efficient distribution of binary patches [Poc] [BM99]. Both approaches, while finding differences between two binaries, are incapable of dealing with aggressive link-time profiling-information-based optimizations and will generate a lot of superfluous information in case register allocation or instruction ordering has changed. A bytecode-centric approach to find sections of similar JAVA-code is studied in [BM98].

Recently another approach to binary comparison also dealing with graph isomorphisms was discussed in [Tod]: Starting from the entry points of an executable basic blocks are matched one-to-one based on instructions present in them. If no matching is possible, a change must have occured. Due to the reliance on comparing actual instructions, a significant number of locations is falsely identified as changed - the paper mentions that about 3-5 % of all instructions change between two versions of the same executable.

## 3 Graph-Centric analysis

Instead of focusing on the concrete assembly level instructions generated by a compiler, we focus on a *graph-centric* analysis, neglecting as much of the assembly as possible and instead analyzing only structural properties of the executable.

### 3.1 An executable as Graph of Graphs

We analyze the executable by regarding it as a *graph of graphs*. This means that our executable consists of a set of functions $F := \{f_1, ..., f_n\}$. They correspond to the disassembly of the functions as defined in the original C sourcecode. The *callgraph* of the program is the directed graph with $\{f_1, \ldots, f_n\}$ as nodes. The edges of this graph represent function calls: An edge from $f_i$ to $f_k$ implies that $f_i$ contains a subfunction call to $f_k$.

Every function $f_i \in F$ itself can be represented as a *control flow graph* (or short *cfg*) consisting of individual basic blocks and their branch relations. Thus one can represent an executable as a *graph of graphs*, e.g. a directed graph (the *callgraph*) in which each node itself corresponds to a *cfg* of the corresponding function.

### 3.2 Control Flow Graphs

The concept discussed here is well-known in literature on compilers and code analysis [AVA99]. Every function in an executable can be treated as a directed graph of special shape. Every node of the graph consists of assembly instructions that imply the execution of the following instruction in memory if and only if the previous instruction in memory was executed. To clarify this: Let $i_k, i_{k+1}$ be addresses of two assembly-level instructions which are adjancent in memory. These instructions belong to the same basic block if the execution of $i_k$ at $n$ steps of execution implies execution of $i_{k+1}$ at $n + 1$ steps, and the execution of $i_{k+1}$ at $n + 1$ implies execution of $i_k$ at step $n$.

Control flow graphs have a few special properties:

1. Every *cfg* has a unique entry point, meaning a unique node that is not linked to by any other node.
2. Every *cfg* has one or more exit points, meaning nodes that do not link to any other node.

Figures 1 through 4 show a simple C function (figure 1), it's assembly-level counterpart (figure 2), a full *cfg* containing the assembly-level instructions (figure 3) and finally just the *cfg* of the function.

### 3.3 Retrieving the information

In order to retrieve these graphs from an executable, a good disassembly of the binary is needed. The industry standard for disassembly is [Dat], mainly due to its excellent cross-platform capabilities coupled with a programming interface that allows retrieval of

```
int foo( int a, int b ) {
  while( a-- ) {
    b++;
  }
  if( b > 100 )
    return 1;
  else
    return 0;
}
```

Figure 1: The C function

the needed information without knowledge of the underlying CPU or its assembly. This facilitates implementing the described algorithms only once but testing them on multiple architectures.

### 3.4 Indirect calls and disassembly problems

In many cases creating a complete callgraph (which represents all possible relations between the different functions) from a binary is not trivial. Specifically indirect subfunction calls through tables (very common for example in C++ code that uses virtual methods) are hard to resolve statically.

In the presented approach, such indirect calls whose targets cannot be resolved statically, are simply ignored and treated as a regular assembly-level instruction. In practice, this does not yield many problems: The question whether a certain call is made directly or not is not answered by the optimizer but by the code that is being compiled, and thus does not change between different builds of the same program without a source code change.

## 4 Structural matching

The general idea explored in this paper is matching the functions in two executables by utilizing both information derived from the *callgraph* and the respective *cfg*'s instead of relying on instructions or instruction patterns. In this section two versions of the same executable will be considered: $A$ and $B$ as well as their callgraphs $\mathfrak{A} := \{\{a_1, \ldots, a_n\}, \{a_1^e, \ldots, a_m^e\}\}$ and $\mathfrak{B} := \{\{b_1, \ldots, b_l\}, \{b_1^e, \ldots, b_k^e\}\}$ which consist of their respective nodes (functions) and edges ($c_i^e$ is a 2-tuple containing two nodes, and thus describes an edge in the graph).

Ideally, we want to create a bijective mapping $p : \{a_1, \ldots, a_n\} \rightarrow \{b_1, \ldots, b_m\}$. In the general case, this mapping does not exist due to different cardinalities of the two sets (if functions have been added or removed). Furthermore, properly embedding $\mathfrak{B}$ into $\mathfrak{A}$ seems to be an excessively expensive operation, specificially considering the possibility of

```
  push    ebp
  mov     ebp, esp
  push    esi
  push    edi
  jmp     short loc_40126A
loc_401267:
  inc     [ebp+arg_4]
loc_40126A:
  mov     edi, [ebp+arg_0]
  mov     esi, edi
  sub     esi, 1
  mov     [ebp+arg_0], esi
  cmp     edi, 0
  jnz     short loc_401267
  cmp     [ebp+arg_4], 64h
  jle     short loc_401287
  mov     eax, 1
  jmp     short loc_40128C
loc_401287:
  mov     eax, 0
loc_40128C:
  pop     edi
  pop     esi
  pop     ebp
  retn
```

Figure 2: The assembly code

$m < n$.

A different iterative approach to creating an approximation of $p$ is taken: An initial mapping $p_1$ is created which maps elements of $A_1 \subset \{a_1, \ldots, a_n\}$ to elements of $B_1 \subset \{b_1, \ldots, b_n\}$. The mapping is then used to iteratively create a sequence of mappings $p_2, \ldots, p_h$ with $A_1 \subset A_2 \subset \cdots \subset A_h$ and $B_1 \subset B_2 \subset \cdots \subset B_h$.

### 4.1 A simple matching heuristic

Comparing undirected graphs is well-known to be excessively expensive, and even the restricted directed graphs can be quite expensive to compare. A relatively simple (and very imprecise) heuristic for telling whether two graphs are isomorphic is to compare the number of nodes and edges. If they do not match, it can be said with certainty that no isomorphism exists. The initial partial mapping $p_1$ is constructed by associating every $b_i \in \{b_1, \ldots, b_m\}$ with a 3-tuple $(\alpha_i, \beta_i, \gamma_i)$ where $\alpha_i$ is the number of basic control blocks in $b_i$, $\beta_i$ is the number of edges in $b_i$ and $\gamma_i$ is the number of edges in the calltree

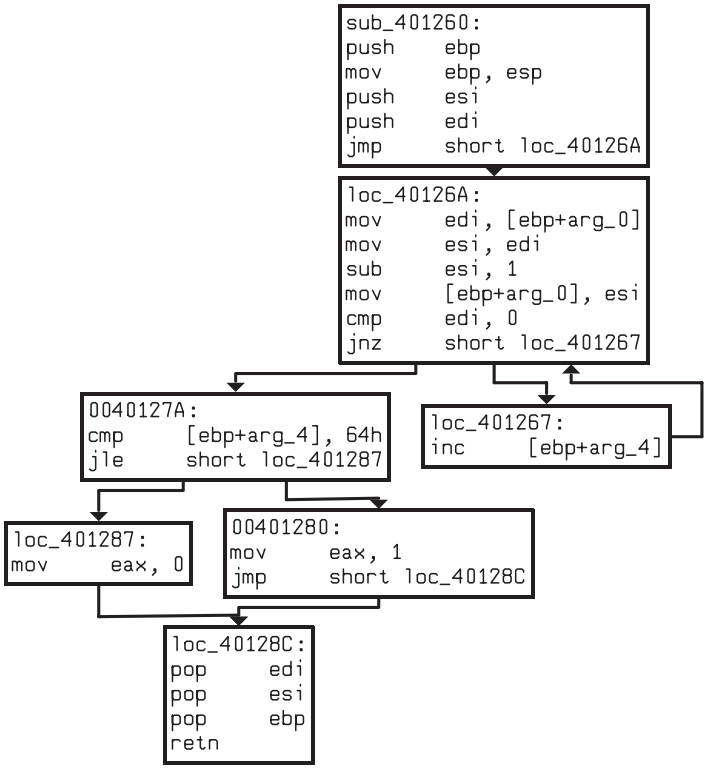


Figure 3: *cfg* with assembly

originating at $b_i$. We denote the mapping that maps a function in a callgraph $\mathfrak{C}$ to it's 3-tuple with $s : \mathfrak{C} \rightarrow \mathbb{N}^3$ and define the inverse function $s^{-1} : \mathbb{N}^3 \rightarrow \mathfrak{P}(\{c_1, \ldots, c_o\})$ that retrieves the set of functions which map to a certain tuple.

The mapping $p_1$ is constructed by examining all 3-tuples generated from $A$ and $B$ as follows: The functions $a_i \in A$ and $b_j \in B$ are mapped to each other if and only if they map to the same tuple and no other element exists in $\{a_1, \ldots, a_n\}$ or $\{b_1, \ldots, b_l\}$ which maps to the same tuple. More formally:

$$p_1(a_i) = b_j \Leftrightarrow |s^{-1}(s(a_i))| = 1 = |s^{-1}(s(b_j))| \wedge s(a_i) = s(b_j) \tag{1}$$

If the cardinalities of the sets $s^{-1}(s(a_i))$ and $s^{-1}(s(b_j))$ are both equal to one and both $a_i$ and $b_j$ map to the same tuple, $p_1$ maps $a_i$ to $b_j$.

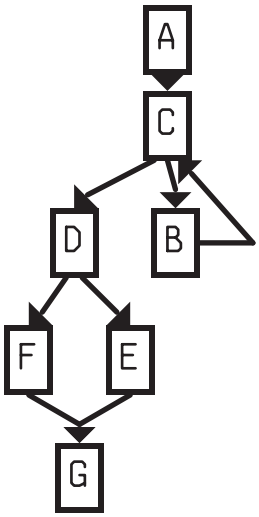


Figure 4: *cfg* without assembly

### 4.2 Improving $p_1$

The above heuristics yield only relatively small subsets of $A := \{a_1, \ldots, a_n\}$ and $B := \{b_1, \ldots, b_l\}$ that can be successfully matched initially. In general, smaller functions are a lot less likely to be successfully matched. This is mainly due to the special form that *cfg*'s take: Most basic blocks have exactly two "children" - this means that the odds that two randomly chosen *cfg*'s with the same node count have the same number of edges decrease as *cfg*'s grow. Smaller functions tend to have fewer subfunction calls, furthermore increasing the likelyhood that $|s^{-1}(s(a_i))| \neq 1$ occurs. It is intuitively clear that smaller sets $A$ and $B$ would reduce the odds of such collisions.

The improved mappings $p_i$ are constructed by taking advantage of the information gained from $p_{i-1}$ and using them to create small subsets $A'_j \subset A$ and $B'_j \subset B$ which are used for improving the mapping as explained above. The algorithm for constructing $p_i$ from $p_{i-1}$ works as follows:

1. Take the $i^{th}$ element $a_i$ from $A_{i-1}$ and retrieve $p_{i-1}(a_i)$
2. Let $A'_i$ be the set of all functions $a_k$ that have edges originating from $a_i$ leading to $a_k$ in $\mathfrak{A}$ and $B'_i$ the set of all functions $b_o$ that have edges originating from $p_{i-1}(a_i)$ leading to $b_o$ in $\mathfrak{B}$
3. Construct $p'_i : A'_i \rightarrow B'_i$ in the same way as $p_1$ was constructed
4. $p_i(a_j) := p_{i-1}(a_j)$ if $a_j \in A_{i-1}$. If $a_j \notin A_{i-1}$ and the construction of $p'_i$ yielded a match, $p_i(a_j) := p'_i(a_j)$. If the construction of $p'_i$ did not yield a match and $a_j \notin A_{i-1}$ then $p_i(a_j)$ is undefined.

5. $A_i$ and $B_i$ are the domain and image of $p_i$

Once $p_k$ has been constructed where $|A_k| = k$, the iteration is finished and cannot yield improved results.

### 4.3 Graph restructuring

Compilers (and optimizing linkers) tend to change *cfg*'s in ways that do not truly change the logical structure of the function. Oftentimes, a single control block in a *cfg* is split up into several smaller ones that are linked with an unconditional branch instruction [1]. Since these operations will change the node and link count a way to easily and quickly undo the changes is needed. A simple graph-restructuring algorithm is applied before generating the 3-tuples which removes superfluous nodes generated by these optimizations:

```
for x ∈ {c_1, ..., c_o}:
    if (number of edges to x) = 1:
        y_x^e := edge to x
        y := source node of y_x^e
        {x_i^e, ..., x_j^e} := set of edges originating in x
        remove edge from y to x from graph
        remove x from graph
        for x^e ∈ {x_i^e, ..., x_j^e} :
            add edge from y to target of x^e to graph
            remove x^e from graph
```

## 5 Practical results

An implementation of the described methods has been created as an extension to the commercial debugger IDA Pro.

### 5.1 Name porting between databases

Two libraries that come with every standard install of Windows were examined in different versions: wininet.dll and msgsvc.dll. The versions of wininet.dll were those of Windows XP SP1/SP2 respectively, the versions of msgsvc.dll those pre/post MS03-48. Both have been heavily fragmented by aggressive link-time optimizations and pose significant problems to signature-based function matching.

[1] This seems to be a specialty of Microsoft's optimizing linker

| File | File Size | # functions | # mapped | Runtime in seconds |
|---|---|---|---|---|
| msgsvc.dll pre MS03-48 | 35.600 | 134 | 100 | $< 5$ |
| msgsvc.dll post MS03-48 | 34.064 | 129 | 100 | $< 5$ |
| wininet.dll SP1 | 599.040 | 2310 | 1522 | 183 |
| wininet.dll SP2 | 588.288 | 2321 | 1522 | 183 |

### 5.2 Analysis of security patches

#### 5.2.1 H.323 Parser

After the NISCC published information about vulnerabilities in multiple H.323 parsers, the question arose where the relevant mistake in Microsofts ISA Server product was. Microsoft refuses to publish detailed information about the vulnerability they fix. According to the NISCC report, the problem was located in ASN.1 decoding.

Both the pre- and post-patch versions of H323ASN1.DLL were analyzed, with the result that 11 functions in the unpatched version could not be mapped to the patched version, and 8 functions in the patched version could not be mapped to any function in the unpatched version.

| Address | # Nodes | # Links | # Children |
|---|---|---|---|
| 40f627 | 26 | 46 | 21 |
| 40f837 | 19 | 32 | 12 |
| 41d012 | 10 | 16 | 7 |
| 41ed06 | 8 | 12 | 2 |
| 428d36 | 8 | 12 | 2 |
| 42b9e2 | 8 | 12 | 2 |
| 42bc90 | 8 | 12 | 2 |
| 42bd85 | 8 | 12 | 2 |

Patched Version

| Address | # Nodes | # Links | # Children |
|---|---|---|---|
| 40f4bb | 22 | 40 | 21 |
| 40f697 | 14 | 24 | 12 |
| 41cd73 | 9 | 15 | 8 |
| 41ce7d | 8 | 13 | 7 |
| 425595 | 4 | 5 | 4 |
| 425728 | 4 | 5 | 4 |
| 428b72 | 7 | 10 | 2 |
| 42b98e | 7 | 10 | 2 |
| 42bbd2 | 7 | 10 | 2 |
| 42bcbf | 7 | 10 | 2 |

Unpatched Version

A manual inspection of these functions yielded the result that the first three functions in both tables are in fact the same with the only change being an added range check. In all three cases, the old version retrieves an unsigned 32-bit integer from an ASN.1 PER encoded stream by means of a function called ASN1PERDecU32Val().

This 32-bit integer is passed on to ASN1PERDecZeroTableCharStringNoAlloc() as second argument. The patched variant on the other hand introduces a range check to make sure this second argument is smaller than 129.

A closer inspection of ASN1PERDecZeroTableCharStringNoAlloc() reveals that the function calculates the size of memory allocation based on the formerly untrusted value – an attacker was able to set this value in a manner that the calculation would exceed

MAXUINT and thus be of very small size. The subsequent copy-operation would then corrupt the heap, allowing an attacker to gain control in the next round of heap consolidation. Instead of fixing the issue at the core (e.g. in the MSASN1.DLL library), a range check was added into the calling application (H323ASN1.DLL).

The update thus disclosed to an examining party that every call to ASN1PERDecZeroTableCharStringNoAlloc() needs to have argument checking done *before* the call is issued. A short system-wide scan was conducted to see if other applications besides ISA Server use the relevant function in dangerous way. Two other instances were found: The Windows-internal H.323 Multimedia Provider Library (which allows arbitrary applications to easily process H.323 data) and Microsoft's Video Conferencing Software Netmeeting. Neither does proper range checking on the function in question.

The result was that the update to H323ASN1.DLL fixed one bug but alerted anyone with the capability to analyze patches to two further remotely exploitable vulnerabilities which were not fixed at the time.

Microsoft was contacted and the issues were fixed a few months later, in MS04-11.

The total analysis took less than 3 hours time.

### 5.2.2 SSL/PCT Parser

In April, Microsoft issued an update to SCHANNEL.DLL, the library responsible for handling SSL communication. According to their security bulletin, they removed a security problem that allowed attackers to take full control of any computer running an SSL-based server. No technical details were provided, except that the problem itself lay in a part of the library responsible for parsing PCT packets [2].

More than 20 changed functions were detected in total, but only one with a name that implied it was involved with PCT parsing. An examination of the function Pct1SrvHandleUniHello() revealed that the old version had taken a string, NOT'ed every character and appended it to the original string. The new version was changed in such a manner that it ensured the combined string would not exceed 32 characters.

Detecting and understanding the vulnerability (a vanilla stack-smash with EIP overwrite) took less than 30 minutes. Subsequently, code was constructed to reach the appropriate location in the binary. Within 5 hours, EIP could be overwritten with an arbitrary value, and within 10 hours of the start of the analysis, a program that reliably exploited the vulnerability was created.

## 6 Comparison to other methods

In comparison to other methods for reverse engineering changes to a binary, the presented method has a few distinctive advantages as well as a few significant disadvantages.

[2] PCT is a legacy-protocol that was obsoleted by TLS and is supported for legacy browsers

### 6.1 Few False positives

The presented method performs significantly better than [Tod] in terms of false positives: The instruction-based approach suffers from 3-5 % of all instructions being marked as changed. Unless heavy, structure-changing optimizations are performed (such as the inlining of complex functions), the presented method is free of false positives: A functions whose flowgraph has changed has undergone a change. While testing the method on a multitude of different programs, no function pair was found that had not changed but was marked as changed. This drastically reduces the human work involved when trying to detect the significant changes in a security update.

### 6.2 CPU-independence

The presented method is almost completely independent of the underlying CPU architecture as long as a good disassembly with cross-references is available. The only CPU-dependent function that has to be available in addition to flow information is the capability to distinguish between a subfunction-call and a non-subfunction call. Successful tests were ran examining differences between MIPS-based ROM images and SPARC-based Solaris ELF executables in addition to the x86-based PE files discussed above.

Instruction-based approaches contain large amounts of CPU-dependent code which makes creating a multi-platform analysis tool significantly more complex.

### 6.3 Possible False Negatives

The downside obviously is the presence of possible false negatives if the program logic itself is not changed but constants or buffer sizes are. It is easy to imagine that a software vendor will fix a security vulnerability not by adding a range check but by enlarging the size of a buffer, which in the current method will go unnoticed. This is where [Tod] is clearly superior, as any change in buffer sizes or constants will be detected. This is bought by the cost of having to examine a significantly larger number of detected changes. Empirical evidence suggests that security updates which change constants but not program flow are very rare. Nonetheless, this is a region in which improvements on the proposed method are desirable.

## 7 Summary

It has been shown that nondisclosure of vulnerability information is not a promising deterrent to would-be-attackers and that security updates can be reverse engineered in relatively little time (given the right tools). It has furthermore been shown that special care has to be

taken when releasing security updates, as the information in the patch has to be assumed to be public. An incomplete bugfix can do more harm than good by disclosing the existence of other (unfixed) bugs along with the fix.

The presented work furthermore implies that the common practice of leaving one or two weeks between the publication of a security update and installing the patch is highly dangerous.

Leaving the politics of vulnerability disclosure out, it has been shown that analysis of binaries based only on structural properties of the code is a promising field of research, as it allows analysis of executable code without the need to abstract to an intermediate language or CPU-specific analysis engines.

## 8 Future work

Many things have to be improved and worked on to make the proposed method truly useful. Fast heuristics that can tell that two graphs are not isomorphic which are better than the current version are needed and would greatly improve the matching statistics. Furthermore, intraprocedural difference analysis would be useful: Given two *cfg*'s $a_1, b_1$ which are different, but expected to belong to the same function due to their position in the callgraph or due to other heuristics, an algorithm that constructs a partial isomorphism between subgraphs of $a_1$ and $b_1$ would allow quicker analysis of changes. Given two versions of the same large function, finding a relatively small change still has to be done manually.

A separation of the function-matching for name porting and function-matching for binary difference analysis will be needed sooner or later: Function-matching benefits from relaxed heuristics, while binary difference analysis does not want to miss changes.

More in-depth study of the effects of heavily optimizing/inlining compilers would be desirable, as well as more studies on the applicability of the presented methods to other CPU-architectures.

Detecting changes in buffer sizes and changes in (certain) constants would be desirable goals in the immediate future.

## 9 Acknowledgements

The author would like to thank the anonymous reviewers for many constructive comments. Valuable comments were also provided by Josh Anderson, Brandon Baker, John Pincus, Felix Lindner und Jan Muenther.

## References

[AVA99] Jeffrey D. Ullmann Alfred V. Aho, Ravi Sethi. *Compilerbau*. Oldenburg Verlag, München Wien, 2 edition, 1999.

[BM98] Brenda S. Baker and Udi Manber. Deducing Similarities in Java Sources from Bytecodes. pages 179–190, 1998.

[BM99] Brenda S.Baker, Udi Manber and Robert Muth. Compressing Differences of Executable Code. In *ACMSIGPLAN Workshop on Compiler Support for System Software (WCSS)*, pages 1–10, 1999.

[Dat] DataRescue. IDA Pro Disassembler *http://www.datarescue.com/idabase*.

[Hir77] Daniel S. Hirschberg. Algoritms for the Longest Common Subsequence Problem. *J. ACM*, 24(4):664–675, 1977.

[HS77] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common susequences. *Commun. ACM*, 20(5):350–353, 1977.

[Poc] Pocket Soft Inc. RTPatch – Software Update Tool *http://www.pocketsoft.com/whitepapers/whitepaper.html*.

[Tod] Todd Sabin. Comparing binaries with graph isomorphisms *http://razor.bindview.com/publish/papers/comparing-binaries.html*.

[ZW99] Scott McFarling Zheng Wang, Ken Pierce. BMAT - A Binary Matching Tool. *2nd ACM Workshop on Feedback-Directed Optimization*, November 1999.

[ZW00] Scott McFarling Zheng Wang, Ken Pierce. BMAT - A Binary Matching Tool for Stale Profile Propagation. *The Journal of Instruction-Level Parallelism (JILP)*, 2, May 2000.